MASTER IN INNOVATION AND RESEARCH IN INFORMATICS
SPECIALIZATION IN ADVANCED COMPUTING
FACULTAT D'INFORMÀTICA DE BARCELONA (FIB)
UNIVERSTIAT POLITÈCNICA DE CATALUNYA (UPC) – BARCELONATECH

IMPROVING BOUNDS ON LARGE INSTANCES OF GRAPH COLORING

Defense date: October 22, 2019

Author: David Sardà
Thesis supervisor: Ken-ichi Kawarabayashi [NII JST ERATO]
Tutor: Javier Larrosa [FIB CS]

Abstract

Graph coloring is one of the most widely known NP hard problems in the field of computer science. Due to its numerous applications, a high variety of solutions to the problem have been explored through the years. Despite this, research on large instances of the problem has been scarce, mainly due to its practical complexity. These solutions deal mainly with heuristics, and in most cases are only able to determine an upper bound on the optimal number of colors, or an interval in which it is contained within. This thesis explores new methods, using both vertex cover and exact graph coloring algorithms in addition to our implementation of the state of the art, to develop a hybrid algorithm that on most instances is able to tighten the bounds or determine the optimal number of colors outright. In addition, we show that given a good enough vertex cover algorithm, a much simpler large graph coloring heuristic algorithm with an efficiency close or better than the state of the art is possible.

Contents

1 Introduction and state of the art

Graph coloring (GC) is a fundamental problem in computer science, explored extensively through the years due to its many practical applications. Some of these are frequency assignment [1], timetabling [2], printed circuit testing [3], crew scheduling [4] and manufacturing [5], among many others.

Graph coloring is defined as, given a graph $G = (V, E)$, being V the set of its vertices, and E the set of its edges, we require to assign for each $v \in V$ a color in $\{1, ..., c\}$, in such a way that the endpoints of every edge get different colors, or technically $\forall\, v_1, v_2 \in V, v_1 \neq v_2$, given the color assignments (v_1, c_i) and (v_2, c_j), if $\exists e \in E$ such that $e = (v_1, v_2)$, then for a valid color assignment, $c_i \neq c_j$.

The objective of the problem is to minimize the number of colors used, as obviously it's trivial to find a solution in any graph if $c = |V|$, as we can just assign a different color to every vertex.

Despite its usefulness, the dificulty of graph coloring is of NP-hardness [10], even to approximate within $n^{1-\epsilon}$ [9]. Because of this, a variety of solutions have been proposed, mostly dealing with small, manageable cases [18] [17] [8].

However, as information technology is improving on recent years, high quantities of large data networks, such as social, biological and information networks, have been appearing more frequently than ever. These real-world networks are larger everyday, and they are commonly sparse with complex structural patterns [11] [12] [13].

Despite this, research on large graph has been scarce. Rossi et al. [19] proposed a way of coloring large graphs with a combination which leverages triangles, triangle-cores and other properties and their combinations, including a recoloring tecnique. Peng et al. detailed a method which partitions a graph into connected components using vertex-cut to then color them respectively [7]. Verma et al. developed an algorithm for GC by successively coloring k-cores with decreasing k values, and also explored clique-finding for large graphs [6]. Lin, Jinkun, et al. [14] proposed a heuristic that used a combination of heuristics, clique finding, and reduction rules.

2 HybridColoring

Our goal is to make an scalable, efficient graph coloring algorithm that could process properly both small and large graphs. In this regard, we propose a hybrid algorithm that combines and repurposes previous state of the art algorithms to come with a solution that's more accurate and works well with any kind of graph. We also believe that the best way to tackle graph coloring in a general way is to use not only one technique but several, to try to rely on each of their strengths. In the following subsections we will describe the main core of the algorithm proposed and all of its components.

2.1 Main Core

The algorithm implemented uses a combination of the state of the art large graph coloring algorithm described in [14], implemented by us fully in C++, and an exact graph coloring algorithm described in [17], used as a blackbox.
We also developed different versions of our algorithm to compare the results obtained, and depending of the version described we also makes use of a vertex cover algorithm described in [16] as a blackbox.
First of all, let's begin by describe the core of our hybrid algorithm, as can be seen in the following pseudocode:

Algorithm 1 HybridColoring

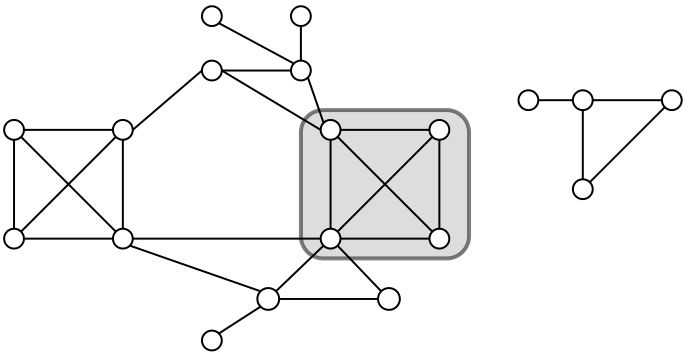
Input: A graph $G = (V, E)$

Output A coloring assignment α and a lower and upper bound lb, ub

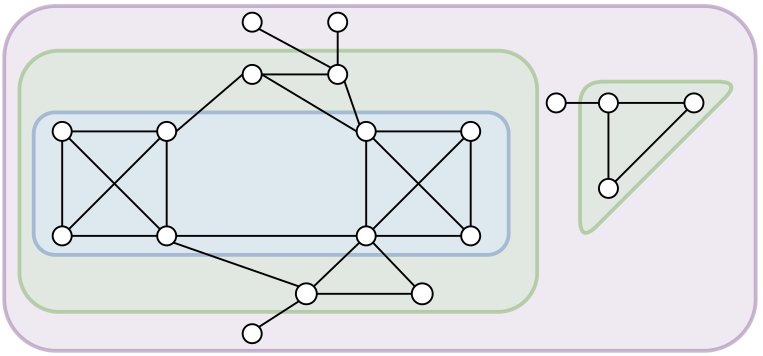
1: $G_k \leftarrow G$
2: $lb \leftarrow 0, ub \leftarrow |V|, ub* \leftarrow |V|$
3: $\alpha \leftarrow \emptyset, \alpha* \leftarrow \emptyset, Cores \leftarrow \emptyset$
4: **for** $i \leftarrow 1$ to c **do**
5: $\quad lb \leftarrow \max(lb, FindClique(G_k))$
6: $[\alpha, Cores, ub] \leftarrow$CoreColorKernel$(G_k)$
7: $[\alpha, lb] \leftarrow$ExactCoreLb$(ub, Cores, G_k, \alpha)$
8: $[\alpha*, ub*] \leftarrow$CalculateColoring$(ub, lb, G_k)$
9: **if** $ub* < ub$ **then**
10: $\quad \alpha \leftarrow \alpha*$
11: $\quad ub \leftarrow ub*$
12: return $[\alpha, lb, ub]$

As we can see, the core of the algorithm is not too extensive because most of the

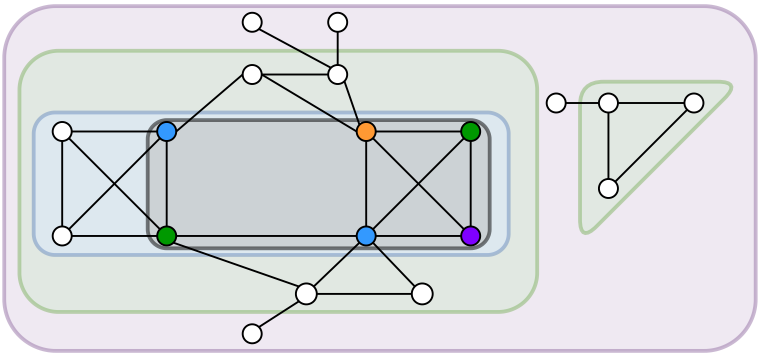
calculations are done by different calls throughout the code. Let's then analyze its functionality. After initialization on lines 1 to 3, we calculate a tentative lower bound on lines 4 to 5 by using the clique finding technique described in [14]. As this function is a heuristic, we call it an arbitrary number of times c (set to 10 on our experiments).
Following that, on line 6 we use part of the graph coloring technique described on the same article to both obtain a first coloring (and hence a new upper bound) and a classification of every node in the graph by different cores (the concept of core decomposition of the nodes in a graph is described in [15]).
Afterwards, on line 7, we use that upper bound, and that core classification to call our own procedure in order to search further for a better lower bound, a technique that is detailed on the next pseudocode on this section. We have to note that a coloring is passed and recieved. This is because, if the graph is small enough after calculating the lower bound, we will just use the exact graph coloring algorihtm to search for the optimal coloring. Finally on lines 8 to 12, we obtain a new coloring by either the saturation based algorithm explained in [14], or a simple vertex cover based algorithm explained on a pseudocode further on this section.
It is to note that all the calculations for the lower bound are performed as early as possible to make the upper bound search more efficient.
Now, let's delve into the main functions that constitute this algorithm. First of all, we will discuss the function that we designed to improve the lower bound of the instance processed. It is based on using the core decomposition of a graph to reach the innermost nodes, and then processing them with an exact graph coloring algorithm to get a tentative lower bound. Obviously this is achievable because if we have a lower bound for a subgraph, we have the same lower bound for the whole graph.



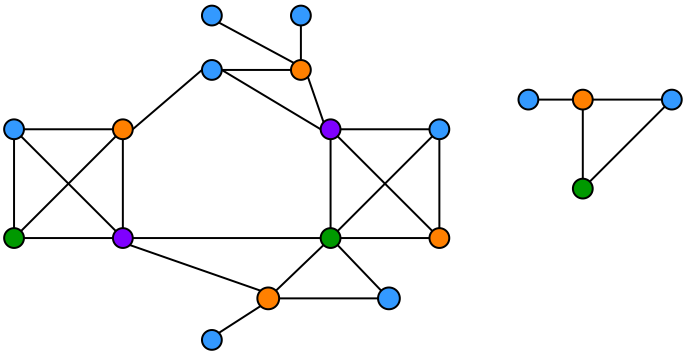
(a) FindClique



(b) CoreDecomposition



(c) ExactCoreLb



(d) CalculateColoring

Figure 1: HybridColoring example

2.2 ExactCoreLb

Algorithm 2 ExactCoreLb

Input: A graph $G = (V, E)$, an upper bound ub, a core decomposition $Cores$, a coloring α

Output A new lower bound lb, a new coloring if the resulting graph is small

```
1: LimitReached ← false
2: CoreSize ← k
3: while not LimitReached do
4:     newLB ← ub
5:     MainCore ← ∅
6:     for i ← 1 to CoreSize do
7:         auxNode ← RandomNode(Cores_MaxLevel)
8:         MainCore ← MainCore ∪ auxNode
9:         Cores ← Cores \ auxNode
10:    Timeout ← false
11:    while not LimitReached and not Timeout do
12:        newLB ← newLB − 1
13:        [LimitReached, Timeout] ←ExactColoring(s, MainCore, newLB)
14:    if Timeout then
15:        CoreSize ← CoreSize − nd
16:    if LimitReached then
17:        newLB ← newLB + 1
18: G ←FilterTrivialNodes(lb)
19: if |V| < minNodes then
20:    LimitReached ← false
21:    α* ← ∅
22:    while Ub > 1 and not LimitReached do
23:        ub ← ub − 1
24:        [α*, LimitReached] ←ExactColoring(G, ub)
25:    α ← α*
26: return [newLB, α]
```

This algorithm uses the one described in [17] to search the innermost core of the graph to try and find a better lower bound. We are interested in finding a coloring with as lower number of colors as possible while obviously being a valid coloring to search for the highest lower bound possible using this method. To mitigate how much time an

exact graph coloring algorithm can spend searching, we will have a time limit on every call we make, and if we surpass it we will iterate again through the main loop, where we will create a smaller subgraph to try and make the search process easier.
Let's go quickly through the lines of the algorithm. After the initializations and entering the main loop, on lines 6 through 9 we construct the subgraph. Following that, we will try and search for the coloring using the exact algorithm on the rest of the main loop, on lines 10 through 17. To do so, we will start by a number of colors equal to the upper bound minus one, as we know the upper bound represents a valid coloring. We will keep trying and decreasing the number of colors until either we find a number of colors which it's not possible to color the subgraph or we time out. If we find this not colorable number of colors we know it increased by one is the new lower bound for the complete graph. If we timeout we just decrease the number of nodes on the subgraph and try again.
Nevertheless, lines 19 through 25 have a different function entirely. Basically, we take advantage of the fact that we have a new (probably better) lower bound and access to an exact graph coloring algorithm to make sure that if the graph is small enough, we just process it with the exact algorithm to then know the optimal number of colors for the general graph, and calculate a valid coloring with it. Finally we return the coloring (which is the old one unless we have calculated it because the graph is small), and a new lower bound.
It has to be noted that on line 18 we filter all trivial nodes. This is a very useful process to simplify the graph. Basically, we eliminate from the graph all the nodes that have a number of edges lower than the lower bound, as they are sure to be able to be colored properly. Then we search for independent sets from the graph on which all its nodes have a lower number of edges than the lower bound. This last process is explained in [14], based on their reduction rule.

2.3 VertexCoverColoring

Let's delve now into the algorithm used to color the entire graph after finishing the lower bound calculation. There are two versions of this algorithm, and either of them achieves the overall coloring of the graph. In this version, we are using a vertex cover algorithm described in [16] to achieve the coloring. The algorithm is simple, the pseudocode is as follows:

Algorithm 3 VertexCoverColoring

Input: A graph $G = (V, E)$, an upper and lower bound lb, ub

Output A new upper bound ub, a new coloring α

1: $G \leftarrow$FilterTrivialNodes(lb)
2: $nodesLeft \leftarrow |V|$
3: $actualColor \leftarrow 1$
4: **while** $nodesLeft > 0$ **do**
5: $\quad Timeout \leftarrow false$
6: $\quad [VertexCover, Timeout] \leftarrow$CalculateVertexCover($G$)
7: $\quad$ **if** not $Timeout$ **then**
8: $\quad\quad [\alpha, nodesLeft] \leftarrow$ColorComplement($VertexCover, actualColor$)
9: $\quad$ **else if**
10: $\quad\quad$ **then**$[\alpha, nodesLeft] \leftarrow$ColorIndependentSetHeuristic($VertexCover, actualColor$)
11: $\quad actualColor \leftarrow actualColor + 1$
12: return $[actualColor - 1, \alpha]$

The basic idea of this algorithm is to achieve the coloring treating every color as a maximal independent set (we will color every node in this independent set with the same color, as no conflicts can arise between the vertices in it).
To find these sets, we will instead find minimal vertex covers, which have direct relation with a maximal independent set. Let's first explain this briefly.
It is well known that the a minimal vertex cover of a graph is the opposite of the maximal independent set of the same graph. By this we mean that a set of vertices $C \subseteq V(G)$ of a graph G is a vertex cover if and only if $V(G) \setminus C$ is an independent set. This is clear, as for every endpoint of an edge, at least one vertex must be in C for C to be a vertex cover, hence not two endpoints of the same edge can be on $V(G) \setminus C$, so then there are no two vertices on $V(G) \setminus C$ connected between them and thus it's an independent set.
It's also clear that as we make C smaller, the vertex cover C will become smaller at the same rate that $V(G) \setminus C$ will become bigger, and thus a minimal vertex cover will lead to a maximal independent set of a graph G.
After understanding this concept the algorithm it's simple, let's go step by step over the pseudocode. On lines 1 through 3 we make the initializations, including the filtering of the trivial nodes in a similar manner as in the ExactCoreLb algorithm. Then, for the rest of the algorithm, we enter a loop where until we have colored all the nodes, we search for an independent set and color it with the highest color yet. For safety we stablish a time limit over the vertex cover algorithm, and if it then takes too long, we use a simple independent set finding heuristic implemented by us. This heuristic is very simple, it

just sorts the remaining nodes by their degrees, and just keeps adding them from lowest degree to highest provided that each new node is not connected to any of the nodes that are already in the set.
Finally, we return the new coloring and the new upper bound.

2.4 SaturationColoring

This is the alternative algorithm that we can use when coloring the graph after the lower bound. It's basically a part of the algorithm proposed on [14], so we will go through it briefly. Its pseudocode is as follows:

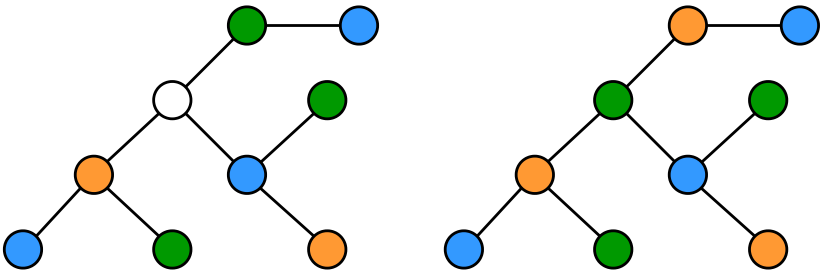
Algorithm 4 SaturationColoring

Input: A graph $G = (V, E)$, an upper and lower bound lb, ub
Output A new upper bound ub, a new coloring α

1: $G \leftarrow$FilterTrivialNodesLayer(lb)
2: **while** $notTimeout$ **do**
3: $B \leftarrow \emptyset, S \leftarrow \emptyset, \alpha \leftarrow \emptyset$
4: $totalColors \leftarrow 1$
5: **for** $i \leftarrow 1$ to $|V|$ **do**
6: $node \leftarrow$random($B_{maxbucket}$)
7: $nodeColor \leftarrow$MinimumColor($node$)
8: **if** $nodeColor > totalColors$ **then**
9: $newColor \leftarrow$Recolor($node$)
10: **if** $newColor \neq -1$ **then**
11: UpdateNeighbors()
12: $nodeColor \leftarrow newColor$
13: **else if**
14: **then** $totalColors \leftarrow totalColors + 1$
15: $\alpha \leftarrow \alpha \cup (Node, nodeColor)$
16: $S \leftarrow$UpdateSaturations()
17: $B \leftarrow$UpdateBuckets()
18: **if** $ub > totalColors$ **then**
19: $ub \leftarrow totalColors$
20: return$[ub, \alpha]$

First of all, it's important to note that this algorithm is designed to work with as much time as it's available, unless of course we have achieved the optimum number of colors of the graph (lower and upper bound meet). Before that though, we filter the nodes in

the same manner as the previous algorithms explained.
So now let's discuss the elements that are needed for this algorithm. The main idea is to use the concept of saturation [18] to design a heuristic to color the graph. Because of this, on line 3, we initialize B, a set of buckets where we will classify every node according to their saturation, and S, a helper set to keep track of every node's saturation. After that, we enter a loop for every node in the graph, in which on every iteration, we will pick at random one of the nodes in the graph that has the highest saturation, and then we will search for its minimum color. If this minimum color surpasses the total colors used to color this graph until now, we make a last attempt to try and prevent it with the recolor procedure [19].
To do so basically we search for a color below total colors that is colliding with only one neighbour, and we search that neighbour's neighbours to see if there is an alternative color available for that other neighbour. If that neighbour can change color this way, we do so and then change the color of our current node to its previous color. As this is a complicated process, we have illustrated it with the following graphical example:



(a) Initial state **(b)** Result

Figure 2: Recolor procedure.

These figures illustrate the simplest example in which recolor can be applied. In this example the middle node, that needs to be colored, cannot be done so if we don't create a new color it. Nevertheless, the green color is only shared by the top left neighbour, and that neighbour can change to another color without conflict, so we just take that color and proceed as described before.
So, after updating both B and S, assigning this new color to the node, and updating $totalColors$, and after each pass updating ub, we just iterate again until a certain, user defined time limit has been met.

3 Experimental Results

The experiments are conducted over a high variety of real world large graphs. We compare them to the state of the art algorithm named FastColor [14] on our own imple-

mentation.
HybridColor and FastColor are both implemented on C++ using the compiler $g++$ version 4.4.7 with the flag -O3. The experiments are performed over a machine using CentOS 6.6 as it's operating system, an Intel Xeon CPU E5-2680 v3 at 2.50GHz and 1TB of RAM (using 700Gb for this project).
For our initialization values on ExactLb we have used a starting size of the core of 120 nodes, and we decrease them by 25 everytime there is a timeout. As for vertex cover coloring, we have a timeout over the vertex cover algorithm of 150 seconds.
The exact coloring algorithm used in ExactLB, and the vertex cover algorithm used in Vertex Cover, are used as blackboxes. The exact coloring algorithm is implemented in C, while the vertex cover algorithm is implemented in Java. For each experiment is listed the number of vertices and edges of the graph, then the lower and higher bound obtained by FastColor, with its execution time in seconds on that instance excluding the reading time of the input file (execution times on parenthesis mean that the algorithm never reaches an optimal, and thus it will never finish until timeout occurs). Then we have the data from HybridColoring divided in the lower bound obtained by ExactLb and its execution time in seconds. Later, we have the upper bound obtained by Vertex Cover coloring, with the execution time in seconds of all the HybridColoring algorithm in total. NºHeur refers to the number of times that a vertex cover was not obtainable due to timeout and it was replaced by the result of our heuristic.

3.1 First experiment set

For our first experiments we went with a variety of real-life networks, including web, biological and infrastructure graphs. On all sets of experimentation we are only interested in the large graphs, as on the small ones we only use the exact coloring algorihtm. We can see the results on the following table:

	|V|	|E|	FastColor			ExactLb		VertexCover		
			LB	UB	ExecTime	LB	ExecTime	UB	TotalExecTime	Nº Heur
NotreDame	325729	1497134	155	155	1.495	117	3.295	155	81.079	0
baidu-relatedpages	415641	3284387	34	95	(1.02)	88	3.425	100	77.583	0
cnr-2000	325557	3216152	84	84	0.213	84	3.072	85	21.79	0
Web-Google	916427	5105039	35	44	(1.047)	44	37.011	44	47.598	0
BerkStan	685230	181043	70	76	(1.017)	76	0.481	78	13.321	0
baidu-internallink	2141300	17794839	29	32	(4.86)	21	57.157	42	4118.307	28
ca-AstroPh	17903	196972	57	57	0.008	57	0.265	57	1165.942	8
ca-CondMat	21363	91286	26	26	0.005	26	90.247	26	244.387	1
ca-dblp-2012	317080	1049866	114	114	0.079	114	1.407	114	15.108	0
socfb-A-anon	3097165	23667394	25	30	(5.914)	25	66.685	44	3746.329	30
socfb-B-anon	2937612	20959854	24	26	(4.328)	24	28.826	40	4076.8541	26
wb-edu	9845725	193822	28	30	(0.997)	28	64.15	30	70.018	0
road-italy	6686493	7013978	3	4	(1.778)	3	9.825	5	7.487	0
road-netherlands	2216688	2441239	3	4	(0.778)	3	3.535	5	0.568	0

Table 1: Results for the web, biological and infrastructure graphs

Let's divide our analysis over the different values. First of all, we can observe an improvement on the lower bound over several of the instances. In web-google, the improvement is such that we can end the algorithm after finding the new lower bound with ExactLb. There are some other instances that have the same lower and upper bound consistently, this is probably due to the existence of a large clique that dominates the coloring of the entire graph. It seems too that on the instance of Notredame, our size limit on our core prevents us on getting the best lower bound.

Moving on to the upper bound, it seems that FastColor still provides overall the best one. Despite this, VertexCover coloring is able to match or at least be close in most cases. The cases where there is the greatest disparity are the ones that VertexCover needed to use the heuristic for its covers in multiple occasions.

Nevertheless, there are cases like ca-AstroPh where Vertex Cover despite using the heuristic several times is still able to match FastColor.

Following that, regarding the execution time, it's clear that FastColor is much better than VertexCover coloring. Despite this, it's important to remember that both the vertex cover algorithm and the exact coloring algorithm are both implemented as blackboxes, meaning that they have to respectively read the input data each time they are called. This means that every cover in the vertex cover algorithm includes a file read, and an appropriate modification of the data by the HybridColoring algorithm. Same thing with the exact coloring algorithm each time the number of nodes and/or the number of colors is different. Is because of this that we don't find this time difference that important.

Finally, looking into the size of the inputs themselves, it seems like overall, while increasing the number of vertices and the number of edges seems to increase the time spent calculating the solution, it's also entirely dependant on a case per case basis.

3.2 Second experiment set

For our second set of experiments we have focused on social network graphs. We can see the results on the following table:

	|V|	|E|	FastColor			ExactLb		VertexCover		
			LB	UB	ExecTime	LB	ExecTime	UB	TotalExecTime	Nº Heur
ca-GrQc	26196	28980	35	44	(0.004)	44	0.106	44	6.353	0
ca-CondMat	108299	186936	19	26	(0.019)	26	90.454	26	214.765	1
wiki-Vote	8297	103689	11	23	(0.263)	19	0.335	40	1408.841	9
ca-HepPh	89208	237010	72	239	(0.861)	120	6.231	239	62.853	0
email-Enron	36691	367662	20	25	(0.172)	20	85.761	40	1541.816	10
ca-astro-Ph	133279	396160	44	57	(0.032)	57	0.583	57	956.151	7
email-EuAll	265213	420045	13	20	(1.381)	18	1.07	36	225.752	1
soc-Epinions1	75887	508837	16	31	(0.619)	25	0.894	48	2823.688	23
soc-Slashdot0811	77359	905468	26	30	(0.048)	28	1.055	45	2020.81	16
soc-Slashdot0902	82167	948464	26	31	(1.066)	29	1.124	48	1992.406	16
ca-dblp-2010	226413	716461	75	75	0.045	75	1.009	75	7.365	0
wiki-Talk	2394384	5021410	17	52	(13.62)	30	68.989	84	5027.219	40
soc-LiveJournal1.txt	4847570	68993773	249	321	(3.469)	125	97.277	340	6259.919	48
ca-hollywood-2009	1069126	56306653	2209	2209	31.637	125	103.726	2209	17660.347	73

Table 2: Results for the social network graphs

The results on this table show that they are consistent throughout the different types of data that we have tested. We can see some more examples of instances where ExactLb improves the lower bound enough to finish the execution, like on Ca-GrQc and ca-astro-Ph.
It also seems that the VertexCover algorithm has a bit more difficulty time-wise on this kinds of graph than the other ones, probably because of the intricacies of social networks in real life. Nevertheless, and maybe because of this, it seems that any time that there is no need to use the heuristic algorithm then the vertex cover coloring algorithm is able to arrive consistently to the same result as FastColor on this set of experiments.
Focusing on the lower bound, it seems that ExactLb is able to improve the lower bound in more than half the cases (10 out of 14). And the two that it didn't, are ones with a lower bound too big to be included in the 125 core graph.

4 Summary and future work

On this thesis we have presented a hybrid algorithm that presents a novel way of calculating both lower and upper bound on the number of colors of graph coloring. It combines state of the art techniques for graph coloring and vertex cover search.
Experiments on this algorithm show that it frequently improves the lower bound, which in some cases leads to finding the optimum number of colors and thus to finishing

the execution right away where it wasn't possible before. Furthermore, using vertex covers this algorithm shows a way to match the state of the art algorithm on most cases experimented with a much more simple approach, though difficulties on its current implementation makes it difficult to determine its full potential.
The results displayed in this thesis further demonstrates the importance of having a middle ground between small and large graph coloring algorithms to provide a needed solution that doesn't rely on any characteristics of the input graphs. Nevertheless it also seems obvious that each technique has its strengths, so we believe that further combining different graph coloring techniques is the way to go. Also, given that the same instance can be explored at the same time by any number of techniques, we believe that parallelizing could lead to further improvements.

Bibliography

[1] Smith, D. H., Hurley, S., & Thiel, S. U. *Improving heuristics for the frequency assignment problem. European Journal of Operational Research*, 107(1), 76-86. 1998.

[2] Burke, Edmund K., et al. *A graph-based hyper heuristic for timetabling problems*. Computer Science Technical Report No. NOTTCS-TR-2004-9, University of Nottingham. 2004.

[3] Garey, M., Johnson, D., & So, H. *An application of graph coloring to printed circuit testing. IEEE Transactions on circuits and systems*, 23(10), 591-599. 1976.

[4] Gamache, M., Hertz, A., & Ouellet, J. O. *A graph coloring model for a feasibility problem in monthly crew scheduling with preferential bidding*. Computers & operations research, 34(8), 2384-2395. 2007.

[5] Glass, C. A. *Bag rationalisation for a food manufacturer*. Journal of the Operational Research Society, 53(5), 544-551. 2002.

[6] Verma, A., Buchanan, A., & Butenko, S. *Solving the maximum clique and vertex coloring problems on very large sparse networks*. INFORMS Journal on computing, 27(1), 164-177. 2015.

[7] Peng, Y., Choi, B., He, B., Zhou, S., Xu, R., & Yu, X. . *Vcolor: A practical vertex-cut based approach for coloring large graphs*. In 2016 IEEE 32nd International Conference on Data Engineering (ICDE) (pp. 97-108). IEEE. 2016, May.

[8] Campêlo, M., Campos, V. A., & Corrêa, R. C. *On the asymmetric representatives formulation for the vertex coloring problem*. Discrete Applied Mathematics, 156(7), 1097-1111. 2008.

[9] Zuckerman, D. *Linear degree extractors and the inapproximability of max clique and chromatic number*. In Proceedings of the thirty-eighth annual ACM symposium on Theory of computing (pp. 681-690). ACM. 2006, May.

[10] M. R Garey, & D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman and Company, San Francisco. 1979.

[11] Barabasi, A. L., & Oltvai, Z. N. *Network biology: understanding the cell's functional organization*. Nature reviews genetics, 5(2), 101. 2004.

[12] Newman, M. E., & Park, J. *Why social networks are different from other types of networks*. Physical review E, 68(3), 036122. 2003.

[13] Adamic, L. A., Lukose, R. M., Puniyani, A. R., & Huberman, B. A. *Search in power-law networks*. Physical review E, 64(4), 046135. 2001.

[14] Lin, J., Cai, S., Luo, C., & Su, K. *A Reduction based Method for Coloring Very Large Graphs*. In IJCAI, pp. 517-523. 2017.

[15] Batagelj, V., & Zaversnik, M. *An O (m) algorithm for cores decomposition of networks*. arXiv preprint cs/0310049. 2003.

[16] Akiba, T., & Iwata, Y. *Branch-and-reduce exponential/FPT algorithms in practice: A case study of vertex cover*. Theoretical Computer Science, 609, 211-225. 2016.

[17] Zhou, Z., Li, C. M., Huang, C., & Xu, R. *An exact algorithm with learning for the graph coloring problem*. Computers & Operations Research, 51, 282-301. 2014.

[18] Brélaz, D. *New methods to color the vertices of a graph*. Communications of the ACM, 22(4), 251-256. 1979.

[19] Rossi, R. A., & Ahmed, N. K. *Coloring large complex networks*. Social Network Analysis and Mining 4.1 (2014): 228. 2014.